A Generic Implementation Approach to Concurrent Fault-Tolerant Software

J. Xu
Dept. of Computer Science
University of Durham
DH1 3LE, UK

B. Randell and A. Romanovsky
Dept. of Computing Science
University of Newcastle upon Tyne
NE1 7RU, UK

Abstract: This paper addresses the practical implementation of means of tolerating residual software faults in complex software systems, especially concurrent and distributed ones. There are several inherent difficulties in implementing such fault-tolerant software systems, including the controlled use of extra redundancy and the mixture of different design concerns. In an attempt to minimize these difficulties, we present a generic implementation approach, composed of a multi-layered reference architecture, a configuration method and an architectural pattern. We evaluate our implementation approach using an industrial control application whose control software we equip with the ability to tolerate a variety of software faults. The preliminary evidence from this case study shows that our approach can simplify the implementation process, reduce repetitive development effort and provide high flexibility through a generic interface for a wide range of fault tolerance schemes.

Key Words — Architectural patterns, concurrent and distributed systems, coordinated atomic (CA) actions, fault-tolerant software, object orientation, safety-critical applications

1 Introduction

This paper investigates the issues concerned with the practical development of fault-tolerant software. The term “software fault tolerance”, in the context of this paper, is concerned with all the techniques necessary to enable a system to tolerate residual software faults, although the effectiveness of these techniques is not usually limited to a precise class of faults. In reality, transient hardware faults, hardware design faults and software bugs often cause similar system behaviour, and measures for coping with software faults can in fact help to deal with some of the other kinds of fault [Powell 1991]. Fault-tolerant software usually involves the introduction of software redundancy to normal program code. The difficulty in finding a non-intrusive way of incorporating redundancy into a complex system often hinders system development and implementation. Furthermore, in many cases the redundancy to be added is application-specific, and the developer has to address both application-dependent and redundancy-related concerns. This complicates further the task of implementing and maintaining realistic fault-tolerant software.

In this paper, we will study how a multi-level reference architecture and a configuration method help to separate different concerns and give application programmers a simple environment for developing fault-tolerant software. We also investigate how an appropriate software pattern provides convenient support for implementing complex fault-tolerant software for concurrent and distributed systems. We reported in FTCS-29 our experience using coordinated atomic (CA) actions as a system structuring tool to design a sophisticated control system for an industrial safety-critical application – The Fault-Tolerant Production Cell [Xu et al 1999]. In the FTCS-29 paper the dependability problems addressed related to faulty sensors, actuators and mechanical devices used by the control system. Here we use the case study again to examine and evaluate our implementation approach, but this time mainly focusing on the problem of possible residual design faults in the control software system.

This paper is organized as follows. Section 2 discusses a general structuring framework for fault-tolerant software. Sections 3 and 4 describe our multi-level architecture and the associated reconfiguration method. Section 5 introduces the GSFT pattern – an architectural pattern for implementing complex fault-tolerant software, especially for concurrent and distributed systems. Section 6 uses the Fault-Tolerant Production Cell case study to illustrate the strengths and weaknesses of our approach, focusing mainly on various types of software faults. Finally, Section 7 concludes the paper.

2 Structuring Framework for Implementing Fault-Tolerant Software

We define a software system as a set of components which interact under the control of a design, and view the components themselves as systems at a lower level of abstraction in their own right [Lee & Anderson 1990]. In order to ease the task of constructing a fault-tolerant software system and control its complexity we believe it is crucial to separate different concerns properly. Our aim is that the users of fault-tolerant components (or FTC Users) should be responsible only for developing their own programs, using services provided by other programmers, without needing much knowledge of how fault tolerance is achieved and implemented. To separate different concerns further, the responsibilities involved in providing such fault tolerance can usefully be divided between:

> *Programmers of Fault-Tolerant Components (or FTC Programmers).* They are responsible for developing components that tolerate certain sets of software faults. They may have to address both functional requirements and fault tolerance aspects. In particular, they must understand different software fault tolerance schemes and be able to develop diverse software variants and application-specific adjudicators. They are required to select and customize an appropriate scheme according to application-specific requirements.
>
> *Programmers of Reusable Control Components (or RCC Programmers).* They are responsible for providing the FTC programmers with a high-level and simple programming interface for the use of the various software fault tolerance schemes. They are also responsible for dealing with low-level implementation details of these schemes.

In order to address the responsibilities of both the FTC and the RCC programmers, we will introduce a multi-level reference architecture for structuring fault-tolerant applications so that different concerns can be addressed properly at separate levels. The major idea behind our

architecture is to hide the control part and low-level implementation details of a fault tolerance scheme from the FTC programmers. This enables the FTC programmers to focus *mainly* on functional requirements and leave the actual implementation of various software fault tolerance schemes to the RCC programmers. However, implementing a concrete scheme in an effective way is never an easy task. Although similar implementation issues have recurred many times in a variety of experimental studies and industrial applications [Lyu 1995], the application programmers, especially novices, still have a hard time understanding and reusing existing solutions. In an attempt to resolve this difficulty, we use pattern techniques [Gamma et al 1995] to document existing and well-proven experience, including our own experience in implementing a generic scheme for software fault tolerance [Xu et al 1995b].

2.1 Idealized Fault-Tolerant Components

An idealized fault-tolerant component [Lee & Anderson 1990] is a (well-defined) component which includes both normal and abnormal responses in the interface between interacting components, in a framework that minimizes the impact of fault tolerance (e.g. extra redundancy) on system complexity (see Figure 1). An interface exception is signalled when the interface checks of the component determine that an invalid service request has been made to the component. A local exception is raised inside the component when the component has detected an error that its own exception handlers should deal with. A failure exception is the means by which the component notifies its caller that it has been unable to provide the service requested of it.

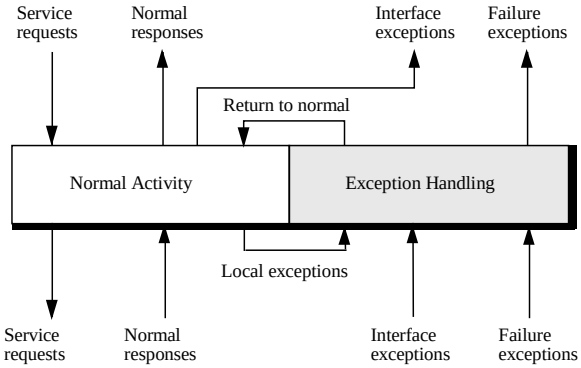

Figure 1 An idealized fault-tolerant component [Lee & Anderson 1990]

We extend this framework to address three further concerns: i) degraded services, ii) concurrency, and iii) embedded software redundancy for tolerating software faults.

Extended interface specification. In practice, when an exception is raised within a component, in many cases the corresponding exception handler cannot deliver the complete service requested by its environment, but possibly only a degraded one. Such responses are conceptually different from both normal responses and failure exceptions, and should be indicated by an appropriate specific exception. It is up to the calling environment to decide how to deal with a degraded service. Similarly, an abort exception should be distinguished from a failure exception. The former notifies the environment that something wrong happened inside the component but all possible effects that would affect the environment have nevertheless been undone, and the latter indicates that it cannot be guaranteed that all

effects have been removed. Figure 2 illustrates the extended interface specification (in which each exceptional response must be indicated by the signalling of an appropriate exception).

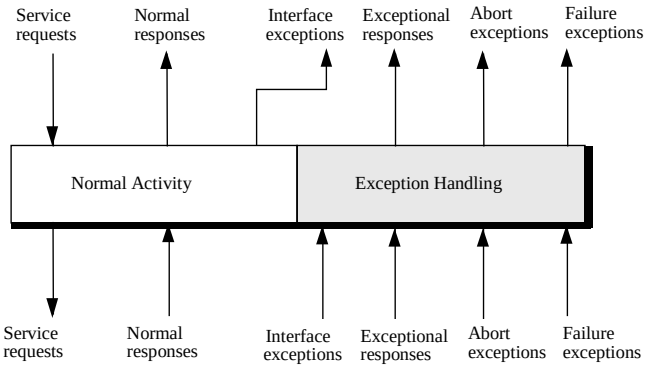

Figure 2 Extended interface of a fault-tolerant component

Interface for concurrent requests. In order to support the development of concurrent and distributed fault-tolerant software, it is often useful to make concurrency explicit at the interface of a component. Linguistically, there are several concrete examples: the multi-function mechanism proposed in [Banâtre et al 1986], multi-party interaction mechanisms [Jung & Smolka 1996], interacting processes [Francez & Forman 1996] and the CA action scheme [Xu et al 1995a]. A component may be associated with *m* synchronous entry points. Service requests may be made concurrently through the entry points by *m* calling components. Within the component, there are exactly *m* roles that execute in parallel in response to service requests and possibly interact with each other to carry out the required computation.

Components with diverse design. A fault-tolerant component can be constructed as a containing component that encloses several diversely designed sub-components called variants and an adjudicator. Variants deliver the same service through independent designs and implementations, and the adjudicator selects a single, presumably correct, result from the results produced by variants. The containing component controls the execution of variants and determines the overall component output with the aid of the adjudicator. This structure is also applicable to a component with *m* synchronous entry points. To provide the same service, each variant must have *m* roles to carry out the required concurrent computation, but may have a diverse internal design for the purpose of software fault tolerance.

2.2 CA Actions and Software Fault Tolerance

A CA action is a mechanism which as well as co-ordinating multi-threaded interactions ensures consistent access to (external) objects in the presence of concurrency and potential faults. CA actions can be regarded as providing a programming discipline for nested multi-threaded transactions [Wing 1993][Caughey et al 1998] that in addition provides very general exception handling facilities. They augment any fault tolerance that is provided by the underlying transaction system by providing means for dealing with i) unmasked hardware and software faults that have been reported to the application level to deal with, and/or ii) application-level failure situations that have to be responded to.

The various approaches to software fault tolerance can be in general divided into two categories: dynamic redundancy and masking redundancy. A system with dynamic

redundancy consists of several redundant components with just one active at a time. If a software error is detected (e.g. by an acceptance test) the active component is replaced by a spare one. A well-known example of dynamic redundancy is the Recovery Block (RB) scheme [Randell 1975]. Masking redundancy uses extra software components (called versions or variants) within a system such that the effects of one or more software errors are masked from the environment of that system. The standard method employed to obtain software fault masking is *N*-Version Programming (NVP) [Avizienis 1985].

Both RB and NVP were developed originally for sequential programs. For concurrent and distributed systems, most reported schemes for achieving software fault tolerance use a process-oriented model and are based on the original idea behind the conversation concept [Randell 1975], i.e. adaptive and coordinated use of a group of recovery blocks (e.g. [Gregory & Knight 1985][Jalote & Campbell 1986]). [Kim & Bacellar 1997] proposed a distributed real time conversation (DRC) scheme with the use of masking redundancy. Here we discuss briefly two schemes for concurrent and distributed object systems by combining the CA action scheme with dynamic redundancy and masking redundancy respectively. For a given CA action *A* and its specification, we will need to develop several variants independently from the same specification. The action *A* can serve as the *container* action, and within the container a set of nested CA actions serve as software variants. They each provide the actual functionality of *A* and together supply redundancy for coping with software faults.

Dynamic Redundancy (DR)

In this scheme, the container action *A* controls the adaptive execution of action variants, which in effect involves the participating threads of *A* performing a sequence of one or more nested CA actions, depending on the errors encountered. Ideally, external objects of container action *A* will be checkpointed for the purpose of action recovery. (Whenever the state restoration of external objects is not feasible, appropriate compensatory operations will be performed instead.)

A container action with three action variants is shown in Figure 3, in which variants are executed in an adaptive manner. If variant 1 ends successfully by passing the acceptance test (or in general the adjudicator), container action *A* will end with a normal outcome. If variant 1 fails to pass the test and thus aborts, variant 2 will be executed after the state of external objects has been restored. Exhaustion of all three variants without success will bring control to the last ditch part. This part can usually deliver some form of exceptional outcome. If even an exceptional outcome is not possible, the original recovery points are used, and an `abort` exception is signalled from action *A* to its surrounding environment. In the worst case that the state restoration cannot be completed, a `failure` exception will be signalled.

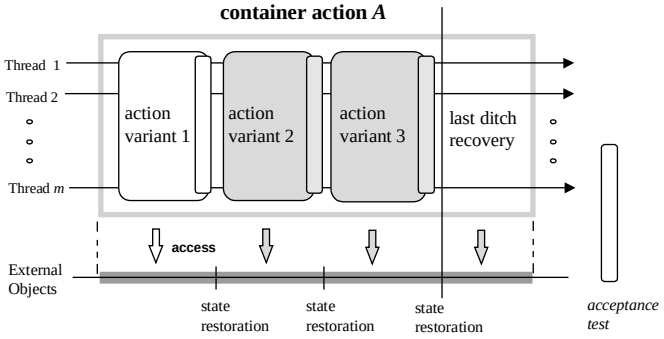

Figure 3 Dynamic Redundancy scheme

Masking Redundancy (MR)

This scheme is essentially based on the concurrent execution of action variants nested in a container action and on determining the majority of the results produced by the variants in order to provide some form of fault masking (see Figure 4). The container action delegates its roles to the nested action variants and afterwards adjudicates the results returned from the variants. Every participating thread of the container action is in effect forked into *n* sub-threads which will participate in *n* action variants respectively. For any external object, *n clones* of it must be made so that *n* action variants can operate on these clones, not on the original object. When all the action variants are complete, for each external object the container action must determine a correct clone by executing an adjudication algorithm. The selected clone will replace the original object and the other clones and the original object will be discarded. This adjudication process in effect merges the sub-threads into the original threads of the container. If the container action aborts for whatever reason, all the clones must be discarded and only the unchanged, original objects retained.

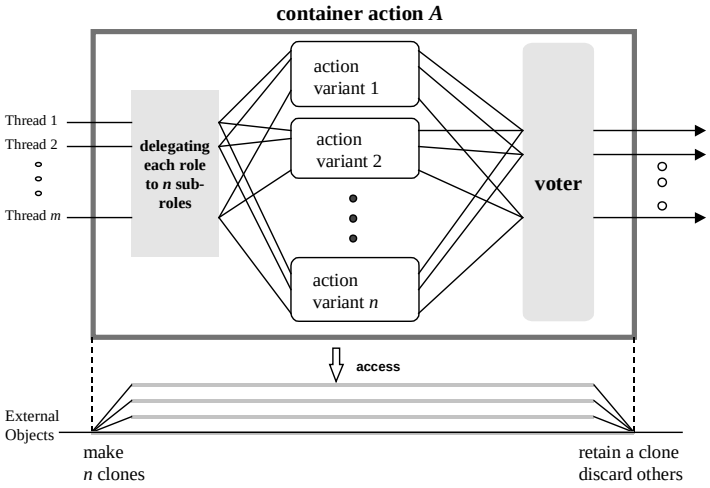

Figure 4 Masking Redundancy scheme

Since every time *n* clones of a given external object must be made for *n* action variants of the container, when objects are large, the process of making clones can be expensive. A possible alternative is to keep *n* copies of an object during the operational time of a system, and so there is no need for cloning the external objects for any individual container action. But it must be ensured that all object copies used are in a consistent state before a container action ends. Any erroneous copies must be corrected by copying the correct state from an unaffected object.

2.3 Development Tools, Support Mechanisms and Architectural Styles

Development tools and support mechanisms for implementing fault-tolerant software can be provided by a variety of techniques such as high-level programming interfaces, libraries of reusable components and development environments, singly or in combination. In practice, it is always desirable that support for implementing fault-tolerant software is widely available and at the same time, if possible, the development of a new language or the modification to an existing compiler is avoided. An approach based on a library of reusable components and tools can free to a great extent the FTC programmers from addressing low-level concerns,

thereby decreasing the complexity of implementing fault-tolerant components. Furthermore, such an approach often exploits object-oriented features, such as inheritance and polymorphism, and needs to use only a limited set of system facilities commonly found in general-purpose operating systems. Without modification of a high-level language or an operating system, rapid and instructive experiments are made possible. (The Arjuna system [Parrington et al 1995] and the ISIS system [Birman 1993] are two successful examples of the use of this approach in the area of fault-tolerant distributed systems.) Here we seek for a more general solution at system-level by developing an architectural style for constructing fault-tolerant applications, employing a library of reusable components. Our solution comprises:

1) A *reference architecture* that consists of multiple levels corresponding to different responsibilities and concerns. For example, at the application level, there might exist both fault-tolerant and non-fault-tolerant components.
2) A simple *configuration method* for selecting and configuring components within the reference architecture to meet particular application requirements. In particular, a configuration method with simple interfaces is provided for the FTC programmers, so that they can customise a specific fault tolerance scheme based on reusable components implemented by the RCC programmers.
3) A library of *reusable components*, which contains reusable experience and expertise in the domain of software fault tolerance. These components may be located at different levels, implementing various control mechanisms and low-level operations. Although different components may have different responsibilities, they have to interact with each other to achieve a global goal such as tolerance to software faults.
4) An *architectural pattern* is used to capture well-proven solutions and to specify precisely relationships between the components and the ways in which they collaborate.

In the following we will discuss our architecture and pattern in an object-oriented fashion though our solutions are equivalently applicable to a traditional functional view of software design. In fact, the object-oriented paradigm fits closely with the idea of idealized fault-tolerant components, and a component can conveniently be thought of as an object [Lee & Anderson 1990].

3 Multi-Level Reference Architecture

Figure 5 illustrates a static view of our reference architecture.

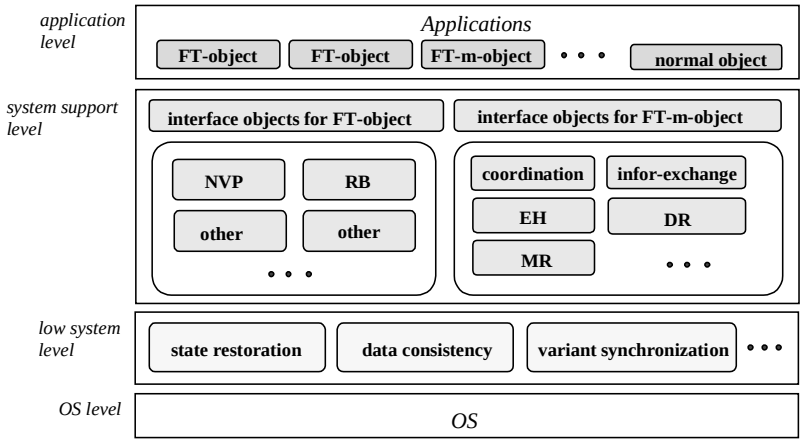

Figure 5 Reference architecture for fault-tolerant applications

This system architecture is composed of several levels: i) the *application level* for the implementation of various applications which may include a set of fault-tolerant objects, ii) the *system support level* composed of interface objects and reusable control mechanisms, iii) the *low system level* that provides low-level mechanisms such as state restoration and variant synchronization, and iv) the *OS level* such as UNIX or Windows NT.

Application Level. This level consists of application-specific objects. Prompted by dependability concerns, some critical objects may be implemented as fault-tolerant objects, and some objects may use or invoke fault-tolerant objects to perform their intended computation. Fault-tolerant objects must adhere to the standard interface characteristics of an idealized fault-tolerant component, as shown previously in Figure 2. Two kinds of fault-tolerant objects are supported: standard ones for sequential invocation and extended ones for concurrent invocation via *m* synchronous entry points, i.e. the FT-m-objects in the figure.

System Support Level. This level provides high-level support for the construction of fault-tolerant objects. There are two special categories of objects that specify interfaces between interacting objects: i) external interfaces and ii) generic FT interfaces. The external interfaces capture application-independent, external characteristics of a fault-tolerant object and specify an abstraction interface between the object and its users. The FTC programmers must follow this structuring framework (e.g. using the inheritance mechanism) when implementing a concrete fault-tolerant object. The generic FT interface objects provide the FTC programmers with a high-level programming interface that facilitates the selection and customization of various fault tolerance schemes. Pre-implemented control mechanisms can include those for RB and NVP and also those for concurrent programs using CA actions, such as coordination of entry and exit, information exchange through external and local objects, exception handling (EH), fault tolerance based on dynamic redundancy (DR) and masking redundancy (MR). More complicated control mechanisms may be implemented at this level by composing the basic control components (see Section 5 for further discussion).

Low System Level. This level offers low-level services which are necessary for certain software fault tolerance schemes, including state saving and restoration for RB, data consistency and variant synchronization for NVP. Other services may be also implemented at this level to provide support for object distribution, concurrency control and reliable communication. Implementation of all objects at the system support level is supported by these services.

OS Level. This level provides conventional OS capabilities. All objects and components at the above three levels may use OS functions directly when the need arises.

4 Configuring Fault-Tolerant Objects

To implement a fault-tolerant object, the FTC programmers are responsible for developing required software variants (and an application-specific adjudicator if needed). With the aid of the reference architecture and any available reusable objects, the FTC programmers can define the fault-tolerant object simply by:

1) reusing (e.g. through inheritance) the standard structuring framework specified by an external interface object to implement an application-specific interface to the users of the fault-tolerant object (i.e. the FTC users), and

2) specifying a software fault tolerance scheme such as RB, DR or MR by selecting the corresponding control mechanism and plugging in the (possibly application-specific) adjudicator, through a generic FT interface.

To request services from a generic FT interface object, the FTC programmers need to pass the references of the variants and the adjudicator to the interface. They may also have to specify the maximum number of processors required for the chosen scheme, and the objects that keep input and output data. The generic FT interface object is also an idealized fault-tolerant component, and so its execution will in general produce either: a normal outcome, an exceptional outcome (signalling a specific exception to the fault-tolerant object), an interface exception, an abort exception, or a failure exception.

5 Architectural Pattern for Implementing Fault-Tolerant Objects

In this section we will introduce a pattern to detail further our solution to the problems that arise in designing and implementing fault-tolerant objects. It extends, to the case of concurrent and distributed systems, the sort of pattern devised by [Daniels et al 1997] based on our original scheme for structuring fault-tolerant software [Xu et al 1995b]. Our solution scheme describes a pre-defined set of reusable classes or objects (located at various levels of the reference architecture), and details their responsibilities and relationships, as well as their cooperation.

5.1 Pattern: Generic Software Fault Tolerance (GSFT)

This GSFT pattern combines the structured characteristics of an idealized fault-tolerant component with the extensibility and flexibility offered by the object-oriented approach. It provides a general way of implementing objects that have the ability to tolerate residual software faults based on a variety of fault tolerance schemes such as RB and NVP. The pattern can be used to implement the fault-tolerant objects that respond to multiple concurrent requests and enclose concurrent activities based on the CA action scheme.

The application objects (or the clients of a fault-tolerant object) address their own functional requirements and request services from the fault-tolerant object through a well-defined interface. A special base class (i.e. an external interface) is defined for fault-tolerant objects from which an application-specific fault-tolerant object can be derived, with inherited interface methods overridden.

The fault-tolerant objects provide dependable services for the clients. They address functional aspects by implementing several software variants for the same functionality and, if needed, a specific adjudication function. The fault-tolerant objects request services from a generic FT interface object and pass the references of the variants and the adjudicator to the interface. The generic FT interface object actually controls the execution of the software variants and adjudicates their results. It allows fault-tolerant objects to specify a particular fault tolerance scheme and, if needed, to change the scheme to another at run-time. This interface object contains a core object called the FT controller from which various schemes can be derived and implemented. Implementing the control mechanism for a special scheme such as RB

requires further low-level services including state saving and restoration, but the FCC programmers should not need to know any implementation details of those low-level services.

5.2 GSFT Structure

The notation used in Figure 6 for inheritance, delegation, aggregation and classes has the usual meaning and is the same as the notation used in [Gamma et al 1995]. (For example, • indicates a subclass relationship, • represents aggregation, — or → shows object reference, and • means many classes.) In Figure 6, the client object invokes services of the fault-tolerant object. The fault-tolerant object is derived from the external interface to conform to the interface characteristics of an idealized component. It requests services from the generic FT interface to execute software variants and the adjudication function.

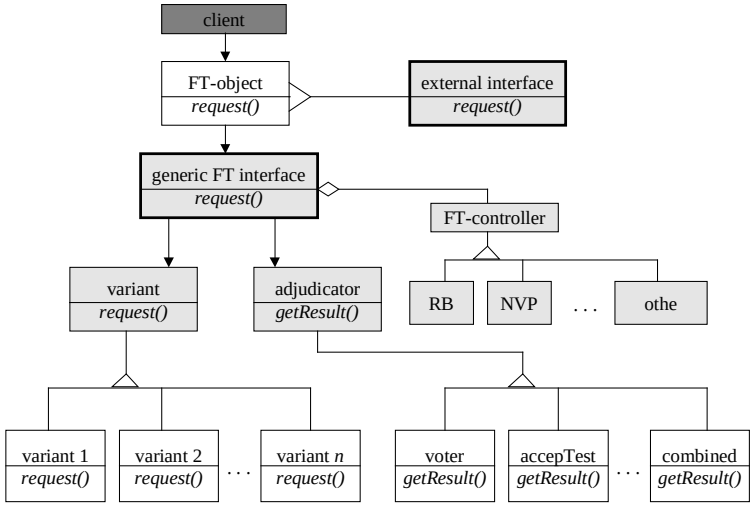

Figure 6 Structure of the GSFT pattern

The generic FT interface object i) requests services from software variants, ii) sends the results of the variant executions to the adjudicator, iii) receives results back from the adjudicator and iv) reports the results back to the fault-tolerant object (which returns the results back to the client in turn). This interface contains an FT controller from which the RB, NVP subclasses etc. can be derived. These subclasses are responsible for actually controlling the execution of software variants and the result adjudication. The variant is an abstract class that declares the common interface for software variants, and the adjudicator is also an abstract class that declares the common interface for adjudication functions. The voter, accepTest (i.e. the acceptance test) and combined (i.e. the combined use of voters and acceptance tests, etc.) can be derived from the adjudicator class to implement actual adjudication schemes.

The structure of Figure 6 is also applicable to concurrent programs. Figure 7 shows a slightly extended structure which provides software fault tolerance based on the CA action scheme. Clients 1, 2, … and *m* are the participants of a CA action. They enter an FT-m-object synchronously by requesting services of the object. The FT-m-object must inherit the standard CA action interface specified by the external interface. It also requests services from the generic FT-m-interface to control the execution of its software variants and adjudication function. The generic interface contains an FT-m-controller from which the EH (concurrent exception handling) [Xu et al 1998], DR and MR subclasses etc. can be derived to implement

a specified fault tolerance scheme. Again, these subclasses are responsible for actually controlling the execution of software variants and performing the corresponding adjudication operation on the results produced by the variants.

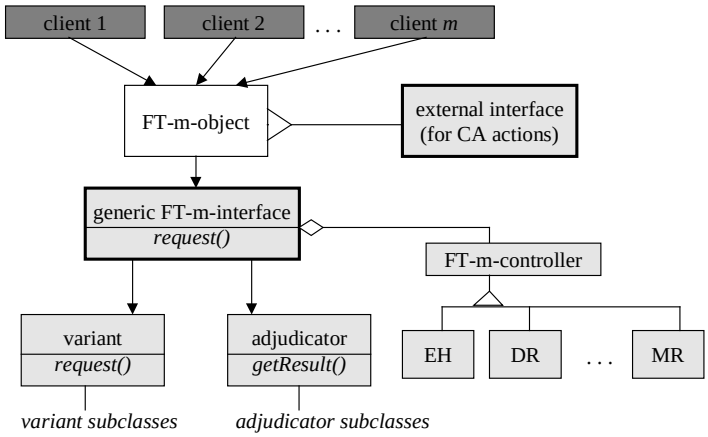

Figure 7 Extended structure for concurrent fault-tolerant software

The CA action scheme is developed for constructing complex concurrent systems and so is significantly more complex than a scheme for sequential programs. To implement the CA action scheme, the FT-m-object requires some additional support including the implementation of cooperative roles, external shared objects and local shared objects. We will discuss these problems and our solutions further when we investigate an actual application example in Section 6. For a particular fault tolerance scheme, certain low-level services may be needed. Take the DR scheme as an example again. State saving and restoration are required. They can be as simple as making a copy of the original object or as complex as recovery cache memory implemented in hardware [Lee et al 1980]. However, the reusable DR object does not have to implement these operations by itself. Instead, it requests a service from the state restoration object located in the low system level to save the system state prior to the execution of a variant and to restore the state if the execution fails to satisfy the acceptance test.

Dynamics

It is difficult to describe the dynamic behaviour of fault-tolerant software systems in general. We present only a typical scenario due to the limitation of space. Figure 8 illustrates the collaboration between objects in the pattern that represents the CA action scheme that uses masking redundancy (MR) to achieve software fault tolerance.

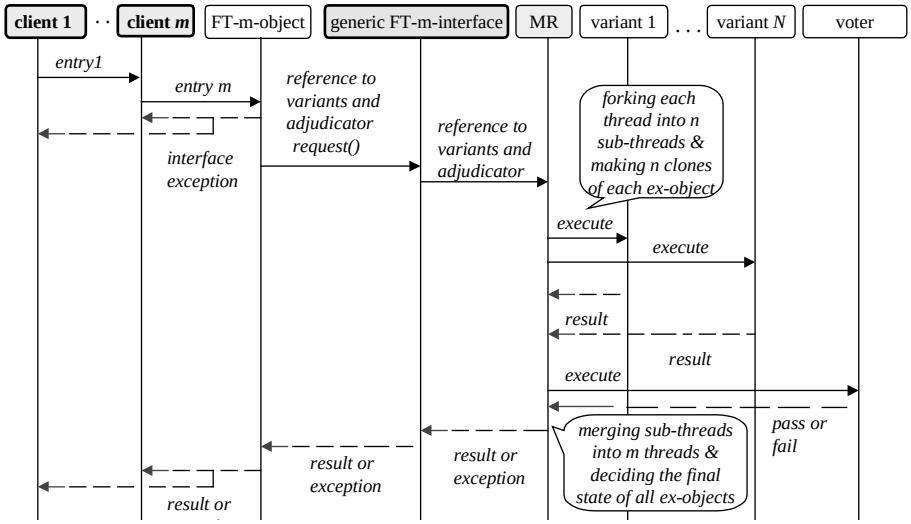

Figure 8 Interaction diagram for CA actions using MR

Multiple clients 1, 2, …, and *m* invoke the services of the FT-m-object through *m* synchronous entry points and pass the references of external shared objects to it. The FT-m-object handles the required service by requesting a service from the generic FT-m-interface. It specifies the chosen scheme as MR and passes the references of external objects, variants and the voter to the interface. Note that both the FT-m-object and the interface component can refuse an invalid service request by signalling an interface exception to their clients. The generic FT-m-interface then delegates the requested service to the MR controller. The controller invokes the execution of *n* functionally equivalent software variants by forking each thread (corresponding to a client) into *n* sub-threads and making *n* clones for each external object. The *m* roles of each variant act upon the corresponding clones of the external objects, and all the clones are forwarded to the voter. The voter attempts to adjudicate these clones and select one of them as the correct result or signal an exception if no majority is found. The MR controller then merges the sub-threads into *m* threads and decides the final state of all external objects. Afterwards the generic FT-m-interface determines its own response to the FT-m-object based on the final state of the external objects. The FT-m-object finally returns an agreed result (i.e. the current state of the external objects) back to the client or signals an appropriate exception.

5.3 GSFT Implementation

We consider here some of most typical issues with the implementation of the pattern, which are applicable to both sequential and concurrent software systems.

System analysis and development. First of all, an appropriate analysis method is used to define a model for the given fault-tolerant application. The services the software should provide, the objects that fulfil these services, and the relationships and collaboration between these objects are identified. Secondly, the model developed in the first step is analyzed to determine which of the application services may request fault-tolerant services and which of the services must be fault-tolerant themselves. The fault-tolerant services that are used to address the dependability aspects of the entire application are defined, and the corresponding fault tolerance schemes that support these services determined. Thirdly, a set of reusable objects that implement control mechanisms for the chosen fault tolerance schemes are defined. The low-level services that support these control mechanisms are then identified.

Finally, a generic FT interface that serves as an external interface to fault-tolerant objects is defined. The fault-tolerant objects can specify a particular scheme, or change from one to another, through the interface. The generic FT interface is responsible for performing all the required changes statically or at run time.

Controlling the execution of software variants: There are several key technical issues we have to deal with carefully when implementing a software fault tolerance scheme. One of them is how to control the execution of software variants. Different implementations exist, including the solutions developed in [Xu et al 1995b], [Rubira & Stroud 1994] and [Tso & Shokri 1996], with different trade-offs between simplicity, generality and flexibility. We provide an alternative object-oriented solution here based on the Composite pattern introduced in [Gamma et al 1995]. This alternative implementation suggests a neat way of using inheritance and aggregation, leading to a simple, but very flexible mechanism for controlling the execution of software variants. The control mechanism is general enough for any fault tolerance scheme using multiple variants. We take the dynamic redundancy (DR) scheme for CA actions as a fairly general example to explain this implementation strategy.

Figure 9 shows the control structure based on the Composite pattern. The DR controls the execution of several variants by sending requests to class variant. The variant class is actually an abstract class that provides a common interface for a set of concrete variants that perform the operations requested. Apart from *n* subclasses implementing software variants, an additional subclass, called Controller, is implemented as an aggregate of those variants and performs the actual control operations. The Controller has to know how many and which variants are needed for a particular request and how they are executed (e.g. sequentially, adaptively, or in parallel). To control the execution of variants, a `Controller` object has to create and store a list of concrete variant objects of its sibling classes.

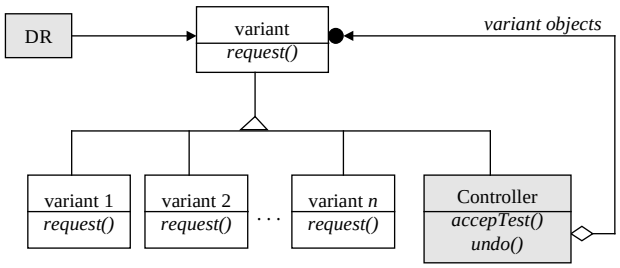

Figure 9 Control structure for CA actions using DR

According to the requests received, a `DR` object decides how many and which variants are needed and then adds the chosen variants to the `Controller` object. For the DR scheme the order in which variants were added to the Controller determines the order in which variants are tried. Because subsequent variants are executed only if their predecessors fail to satisfy the acceptance test, there is a need to maintain information about which variants have failed and which ones should be tried next. This can be done by maintaining the state of the current variant being executed, e.g. using a variable called `currVariant`. The `Controller` constructor has to initialize a list of variants. A pointer to the current variant is maintained in the `Controller`. Initially it points to the first member of the list. The variants in the list will be executed in order until one passes the acceptance test.

State restoration. The GSFT pattern implements state restoration operations as low-level services, which are transparent to the FTC programmers. The implementation of these

operations can use the Memento pattern for checkpointing in [Gamma et al 1995] to save and restore the original state of the objects affected by the execution of the current variant. The operation may have to restore the state of the system, e.g. values of variables for the RB scheme or restore the state of all the external objects for a given CA action.

Adjudication functions. Two basic adjudication functions are the acceptance test for RB and DR and voting for NVP and MR. They can be used singly or in combination. Similar to the reliable hybrid pattern [Daniels et al 1997], we can use the Composite pattern introduced in [Gamma et al 1995] to recursively combine various adjudication functions. Such combined instantiations permit a wide range of adjudication strategies, from the very simplest to highly complex solutions.

Reflective implementation. It is relatively straightforward to implement the GSFT pattern based on inheritance and delegation. However, our experience shows that a reflective implementation is also feasible when using a similar multi-level reference architecture. The base level and meta level can be treated as two separate levels (e.g. application level and system support level) in our reference architecture, each of which provides its own interface. For example, the base level specifies the user interface for exploiting application functionality, and the meta level defines the interface and objects that determine the fault-tolerant behaviour of the application. In general, a reflective implementation can provide a relatively greater degree of transparency than an implementation based on inheritance and delegation. It supports the change of fault tolerance schemes, statically or dynamically, even without modifying any source code of the fault-tolerant objects that use the schemes. However, the reflective implementation is less efficient because of the complex relationship between the base level and the meta level. Reflective capabilities often require extra processing including information retrieval, changing metaobjects, consistency checking, and communication between the two levels. All of these impose some performance penalty.

6 Case Study: The Fault-Tolerant Production Cell

Our reference architecture and the GSFT pattern help determine the basic structure of the solution to the problem of building fault-tolerant software, but they do not specify a fully-detailed solution. They provide a scheme for a generic solution to a family of problems, rather than a prefabricated module that can be used “as is”. This scheme must be implemented according to the specific needs of a particular application in hand. In this section, we use a realistic case study to investigate various details specific to a concrete implementation. The safety-critical application we used in our FTCS-29 paper to study hardware and environmental faults included a control software system. Although the CA action structure has greatly simplified the control software, to the point where many aspects of it have been formally validated [Xu et al 1999], it nevertheless remains rather complex, and the sort of software for which software fault tolerance might be appropriate, and hence makes an interesting example to use to demonstrate our ideas. Indeed it did in fact turn out that, despite the validation efforts there were some residual software faults, and these were coped with well by the fault tolerance we introduced.

An industrial production cell model, based on a metal-processing plant in Karlsruhe, Germany, was first created by the FZI (Forschungszentrum Informatik) in 1993 [Lewerentz & Lindner 1995], within the German Korso Project, in order to evaluate different formal methods and to explore their practicability for industrial applications. In 1996, the FZI

presented the specification of an extended model of the original production cell, called the "Fault-Tolerant Production Cell" [Lötzbeyer 1996]. This second model has additional devices, sensors and warning light systems to facilitate hardware failure detection and fault tolerance.

The Fault-Tolerant Production Cell consists of six devices: two conveyor belts (a feed belt and a deposit belt), an elevating rotary table, two presses and a rotary robot that has two orthogonal extendible arms equipped with electromagnets (see Figure 10). These devices are associated with a set of sensors that provide useful information to a control system and a set of actuators via which the control system can exercise control over the whole cell. The task of the cell is to get a metal blank from its "environment" via the feed belt, transform it into a forged plate by using a press, and return it afterwards to the environment via the deposit belt. Figure 10 illustrates a group of CA actions that constitutes a control system controling the cell. The main characteristics of our control system are the way it separates safety, functionality, and efficiency concerns among a set of CA actions, which thus can be designed, and validated, independently of each other, and of the set of device/sensor-controllers that dynamically determine the order in which the CA actions are executed at run-time. In particular, the safety requirements are satisfied at the level of CA actions, while the other requirements are met by the device/sensor-controllers [Zorzo et al 1999].

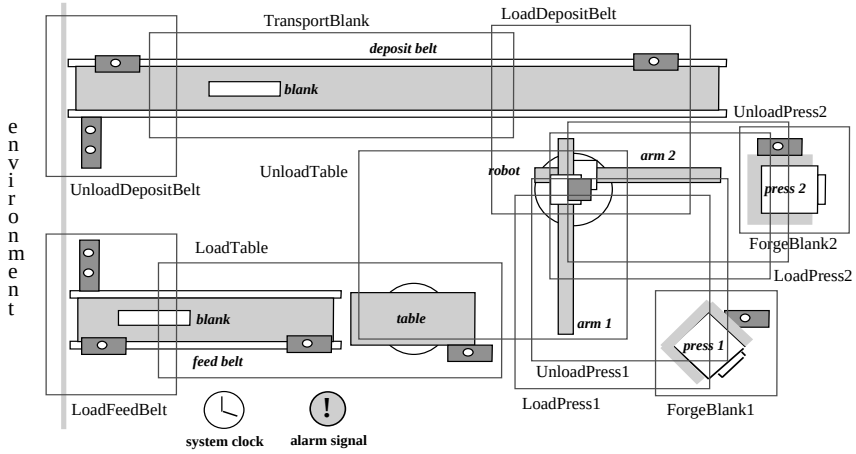

Figure 10 CA actions that control the Fault-Tolerant Production Cell

6.1 Software Faults in the Control Program

A control program for the Fault-Tolerant Production Cell is a complex program with multiple concurrent control threads that must coordinate the interaction of multiple concurrent devices in a highly safe manner. It cannot be simply assumed that the control software itself will be free of error. Software faults might manifest themselves during the system operation time.

Apart from possible incomplete or inconsistent specifications of computation requirements, there are two other sources which may introduce software bugs into the control program. One source is the selection of inadequate or insufficient algorithms which do not cover all possible application situations, especially rare but possible system conditions. Another source is mistakes in converting a selected algorithm into a program for a specific environment such as the controlled cell. In addition, since our control program involves concurrency and/or distribution, software faults can be further introduced into the part that coordinates multiple

concurrent objects. And though individual objects may meet their specifications, the specifications for cooperation between objects are often incomplete and inaccurate.

Software fault avoidance is particularly difficult in realistic safety-critical applications. Although the use of formal methods is very helpful in improving the software quality, formal verification of designs and implemented programs has not yet advanced to the level where the absence of any error in sizeable software can be fully verified by machine.

6.2 Dealing with Software Faults in the Control Program

For simplicity's sake, we investigate here situations involving software faults only, i.e. we assume that *only faults that can occur are software faults in the control program, and all the hardware components of the Production Cell are fault-free*. (Hardware component failures of the cell were discussed in detail in [Xu et al 1999]; we do not anticipate significant problems in providing a solution that exhibits both software and this type of hardware fault tolerance.)

The major software faults that might remain in the control program include i) interaction faults that occur when the interaction relationship between roles of a CA action, between interacting CA actions, or between the control program and the Production Cell has not been specified properly or analyzed sufficiently, and ii) timing faults that occur when an operation, a role or a CA action is not completed in a pre-specified amount of time. Software faults may be also involved in error detection and recovery mechanisms for CA actions. We regard these mechanisms as part of a CA action support mechanism, and assume that the support mechanism itself has been tested extensively and is fault-free. To provide support for this assumption, these mechanisms are implemented as reusable and well-tested components in our GSFT pattern. The CA action support mechanism also contains a concurrency-control mechanism for controlling access to external objects. In the Fault-Tolerant Production Cell, the external objects (i.e. metal blanks) cannot be shared by two or more concurrent CA actions due to safety-related concerns. We choose to use a simple concurrency-control mechanism to minimize the probability of residual software bugs. This mechanism allows a monitoring object to get the state information of a blank concurrently with a running CA action.

Software faults such as interaction faults and timing faults have not been addressed adequately in the initial requirements for the Fault-Tolerant Production Cell [Lötzbeyer 1996], while the high-level specification for implementing the control program (such as the COALA specification used in [Xu et al 1999]) focused mainly on hardware device, sensor and actuator failures. For these reasons, and the inadequacy of available fault removal techniques, we had to admit that despite our best efforts to the contrary, software faults might exist in our control program. This indeed turned out to be the case since we have observed subsequently that some software design faults occur while our control program is in use. For example, a transient software fault occurs in the FZI simulator when arm 1 of the robot is required to place the blank into an unoccupied press. The arm performs most specified operations but it fails to drop the blank into the press. This fault manifests itself only occasionally and makes its removal extremely difficult. Another software bug appears in our control program in the form of an interaction fault. This interaction fault manifests itself only when more than two blanks are placed into the system. Under certain conditions at run-time, two interacting CA actions can be involved in a deadlock situation from which no further operations are possible.

It is therefore evident that it is worth trying to deal with residual software faults that may manifest themselves while the Production Cell is in operation. There are a variety of software fault tolerance schemes we may consider. For transient software faults, a simple "re-try" strategy is often effective [Gray 1990]. For example, the `LoadPress1` action (see Figure 11) may have dropped a blank into press 1 properly, but the nested action `RetractArm1` failed to retract the arm to a correct position. Instead of attempting to execute an action variant, the recovery operation can simply re-execute the nested action in the hope that the same error may not recur. If the control program is executed in a distributed environment, the nested action in question may be re-executed on a different node of the distributed system. Timing faults can be treated by a simple timeout mechanism associated with the CA action support mechanism. For a given task (e.g. the execution of a role or an action) we specify a pre-defined amount of time. If the task is not completed in the time, a timeout exception will be raised. Since there are no strict timing constraints in the initial requirements for the Fault-Tolerant Production Cell, the recovery measure can be quite simple, such as re-execution and abortion. ([Lötzbeyer & Mühlfeld 1996] also introduced a Real-Time Production Cell model for which we have developed a scheme for dealing with exceptions in both the time and value domain [Romanovsky et al 1998].)

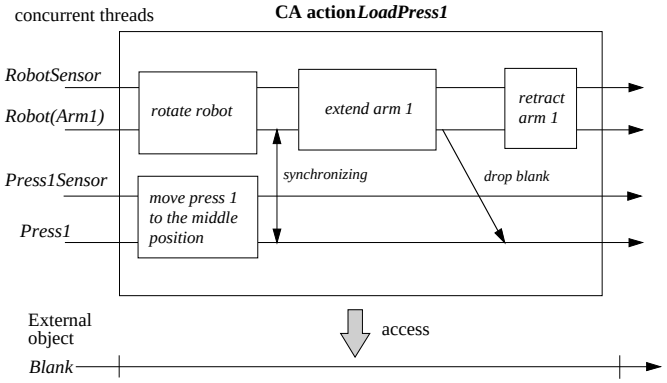

Figure 11 CA action `LoadPress1` and its nested actions

However, for most software design faults (e.g. the interaction fault in our control program) a rollback and re-try approach is insufficient. Instead, tolerance of such software faults must rely on the application of design diversity. We wanted to use design diversity to minimize the probability that the independently designed variants contain similar errors that can cause the variants to fail simultaneously. (Although it is possible introduce diversity into the specification and other phases of the system life cycle, in this case study we will focus on the use of diversity in the design phase and the implementation phase.) While random diversity may be achieved by different programmers and designers, we feel that deliberately-chosen diverse data structures and algorithms are less likely to fail simultaneously. Such an approach is often called "enforced diversity". For example, based on our CA action-based design of the control program, two software variants for the `LoadPress1` CA action were developed using enforced diversity. Variant One is implemented in the form shown in Figure 11, which involves several concurrent activities and two concurrent nested actions `RotateRobot` and `MovePress1toMiddle`. Variant Two is designed to provide the identical functionality but following a simpler algorithm without any concurrency. Within the second variant, five nested CA actions are executed sequentially in the order of `MovePress1toMiddle`, `RotateRobot`, `ExtendArm1`, `DropBlank`, and `RetractArm1`. Because the interaction

relationship between these nested CA actions is essentially diverse in two variants, the probability that the variants fail identically should be reasonably low.

We decided to use the dynamic redundancy (DR) scheme for several practical reasons. Generally speaking, the DR scheme introduces relatively low additional complexity without requiring clones of the external objects and replicating threads. It uses an acceptance test to validate a single result at a time without having to examine all the results produced by multiple action variants. However, we have to keep the test reasonably simple and reduce the probability that the test itself contains software faults. We had previously developed sets of pre- and post-conditions for each main CA action. All the required acceptance tests for CA actions were then derived from the corresponding post-conditions. The correctness of these tests were also validated when our design was examined by extensive formal verification and model-checking. The error-detection coverage is further improved by a large number of executable assertion statements within CA actions and run-time checks supported by the hardware platform. Some of these assertion statements could be switched off when the system performance has to be increased.

Figure 12 illustrates the control structure of the `LoadPress1` CA action using the DR scheme to tolerate software faults. CA action `LoadPress1` is designed as a container action which contains two diversely designed variants. Normally, the container action just executes the first variant. If no error is detected, then the container action ends with a normal outcome. However, if an error is detected by either an assertion statement or the acceptance test (at the end of the first variant), this error must be reported to the container action.

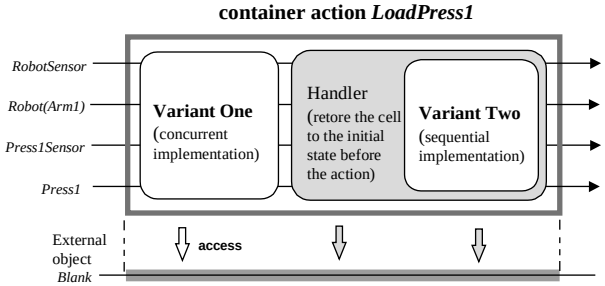

Figure 12 `LoadPress1` as container action to tolerate software faults

In principle, error propagation and error recovery should be performed within a structuring framework for exception handling. (A structuring approach based on exception handling has proved extremely effective in dealing with sensor/actuator faults and faulty mechanical devices in our FTCS-29 paper.) When an error occurs in Variant One, an appropriate exception must be signalled from the variant to the container. The corresponding handler will then be called, which has to restore the related device objects and the blank object to their original state — the state before the execution of action `LoadPress1`. After finishing the restoration, the handler will invoke the second variant in the hope that the same software error will not occur again. In the worst case that an error (not necessarily the same one) takes place again, it is always possible for action `LoadPress1` to signal an `abort` exception, if the state restoration is successful, or a `failure` exception to its containing action. By way of example, we consider the arm 1 position error again. A handler must handle the exceptional situation where the nested action `RetractArm1` fails to place the arm in a correct position.

This handler can use both a re-try strategy and a diversely designed variant for the action. We outline the basic functionality of the handler as follows.

> *Handler for Arm 1 Position Error:* Re-execute the `RetractArm1` action and check the arm position. If the same error persists even after a pre-determined number of re-executions, then restore both the device objects (i.e. `Robot` and `Press1`) and the blank object to their original state. Invoke the second variant. If the variant signals an exception, instead of a normal outcome, then restore the state again. If the state restoration is successful, signal an `abort` exception to the container action, otherwise signal a `failure` exception.

Our implementation of the control program uses 12 main CA actions to address the safety requirements. (Each action concerns a situation in which two physical devices, e.g. the robot arm and the press, have to coordinate their activities.) The application of software fault tolerance techniques at this level enhances the ability of the control program to handle software errors so as to improve both system reliability and safety. Our design also establishes a set of device/sensor-controllers on top of these CA actions to coordinate the execution order of the main actions. Conceptually, this device/sensor-controller level may be considered as a special, systemwide CA action. This outermost CA action starts when the control system begins, and it ends when the system stops normally according to user requirements. To achieve software fault tolerance at the level of the overall control system, all the schemes for sequential programs including recovery blocks and *N*-version programming can be applied directly by treating the outmost action as a single sequential system that encloses complex concurrent activities inside itself. Since we are concerned mainly with safety aspects in this case study, which have been addressed at the level of 12 main CA actions, we will not discuss further details of obtaining software fault tolerance at the level of the outermost system.

6.3 Implementation of the Control Program

We chose Java as the implementation language and used an object-oriented solution based on inheritance and delegation (rather than a reflective architecture). It is evident that the GSFT pattern enables reuse of reusable support mechanisms and simplifies to a great extent the development of a fault-tolerant object by separating different concerns. However, the GSFT pattern provides no support for addressing functional aspects of a CA action. While a pattern captures key properties of an architectural solution, many implementation details are suppressed. We feel that patterns should be treated as just one of many important tools in a toolkit of supporting software development; they cannot free developers completely from necessary analysis, design and implementation issues.

During the testing phase and the demonstration of our implementation, the control system proved highly robust when dealing with hardware-related faults. It is however difficult to determine precisely the ability of the control program to tolerate software faults although we know that the ability depends mainly upon the error-detection coverage provided by a combination of the acceptance test, executable assertion statements and run-time checks by the hardware system. We found that introducing a set of meaningful software faults into the control system that could pass tests and checks is not an easy task at all. Nevertheless, some events that occurred at run-time provided quite encouraging feedback. For example, a previously unknown software fault remaining in the FZI simulator itself was detected

successfully by the acceptance test of a CA action and recovered by the re-try operation associated with that action. This shows, albeit indirectly, that if a similar software error occurred inside the CA action, it would have been caught by the acceptance test and would have been tolerated successfully by a second action variant. With the aid of the GSFT patterns and reusable mechanisms, the additional cost of achieving software fault tolerance was contributed mainly by the cost of developing action variants. We are now in the stage of collecting experimental data for further dependability and performance-related evaluation.

7 Conclusions

The design and implementation of fault-tolerant software for critical computer applications are complex and error-prone tasks – they need an architectural solution that separates different concerns and makes certain aspects transparent to a given type of programmers. We have developed a multi-level reference architecture for implementing fault-tolerant software, which separates the application-specific functionality, interfaces to fault tolerance schemes and application-independent control mechanisms. Such separation has helped us to promote better understanding of both functional and non-functional aspects of a fault-tolerant application and might have resulted in increased dependability of the real application.

Our approach provides support for implementing fault-tolerant software using pre-defined classes and run-time libraries. In principle, it does not require special pre-processors or builders such as the architecture introduced in [Ancona et al 1990] or a new programming language with particular syntax for specifying fault tolerance schemes. Since different groups of objects are located at different levels and low-level services and implementation-details are hidden from higher-level objects, our reference architecture may readily be ported to different platforms, without requiring any direct support from a special underlying operating system. Huang and Kintala of AT&T Bell Labs [Huang & Kintala 1993] developed a library-based approach to checkpointing and backward error recovery. They introduced three reusable components in C that provide fault tolerance in the application layer. Our solution is much more general and can be used as a unifying architecture for implementing a wide range of software fault tolerance schemes for both sequential and concurrent programs.

Although there are some application-specific solutions to the implementation of fault-tolerant software, it remains difficult to reuse fault-tolerant software components directly for complex applications due to the growing heterogeneity of hardware and software architectures and the increasing diversity of operating system platforms. Architectural patterns express the structure and collaboration of participating components at a level of abstraction still higher than source code, thereby facilitating reuse of software architecture, even when other forms of reuse are infeasible. We have found that pattern techniques are very promising with regard to promoting widespread reuse of our architectural solutions for implementing complex fault-tolerant software. We have used the GSFT pattern to equip a control program for the Fault-Tolerant Production Cell with the ability to cope with software faults. The GSFT pattern, together with the object-oriented structure and high-level control abstractions like CA actions, helps a lot in reducing the development effort, simplifying the implementation process and permitting a high level of flexibility and extensibility.

Acknowledgements

This work was supported by the ESPRIT Long Term Research Project 20072 on “Design for Validation”.

References

[Ancona et al 1990] M. Ancona, G. Dodero, V. Gianuzzi, A. Clematis, and E.B. Fernandez, "A system architecture for fault tolerance in concurrent software," *IEEE Comput.*, vol. 23, no. 10, pp.23-32, 1990.

[Avizienis 1985] A. Avizienis, "The *N*-version approach to fault-tolerant software," *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 12, pp.1491-1501, 1985.

[Banâtre et al 1986] J.P. Banâtre, M. Banâtre and F. Ployette, "The concept of multi-functions: a general structuring tool for distributed operating systems," in *6th Int. Conf. Distrib. Comput. Syst.*, pp.478-485, 1986.

[Birman 1993] K. Birman, "The process group approach to reliable computing," *Communications of the ACM*, vol. 36, no. 12, pp.37-53, 1993.

[Caughey et al 1998] S.J. Caughey, M.C. Little and S.K. Shrivastava. "Checked transactions in an asynchronous message passing environment," in *1st Int. Symp. Object-Oriented Real-Time Distrib. Comput.*, pp.222-229, Kyoto, April 1998.

[Daniels et al 1997] F. Daniels, K. Kim and M.A. Vouk, "The reliable hybrid pattern: a generalized software fault tolerant design pattern," in *Int. Conf. PloP'97*, pp.1-9, 1997.

[Francez & Forman 1996] N. Francez and I.R. Forman. *Interacting Processes: a multiparty approach to coordinated distributed programming*, Addison-Wesley, 1996.

[Gamma et al 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Objected-Oriented Software*, Addison Wesley, 1995.

[Gray 1990] J.N. Gray, "A census of Tandem system availability between 1985 and 1990," *IEEE Trans. Reliability*, vol. 39, no. 4, pp.409-418, 1990.

[Gregory & Knight 1985] S.T. Gregory and J.C. Knight, "A new linguistic approach to backward error recovery," in *15th Int. Symp. Fault-Tolerant Comput.*, pp.404-409, Michigan, 1985.

[Huang & Kintala 1993] Y. Huang and C.M.R. Kintala, "Software implemented fault tolerance: Technologies and experience," in *23rd Int. Symp. Fault Tolerant Comput.*, pp. 2-9, Toulouse, 1993.

[Jalote & Campbell 1986] P. Jalote and R.H. Campbell, "Atomic actions for software fault tolerance using CSP," *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 1, pp.59-68, 1986.

[Jung & Smolka 1996] Y-J. Joung and S. A. Smolka, "A comprehensive study of the complexity of multiparty interaction," *Journal of the ACM*, vol. 43, no. 1, pp.75-115, 1996.

[Kim & Bacellar 1997] K.H. Kim and L. Bacellar, "Time-bounded cooperative recovery with distributed real-time conversation scheme," in *IEEE WORDS'97*, 1997.

[Lee & Anderson 1990] P.A. Lee and T. Anderson. *Fault Tolerance: principles and practice*, Second Edition, Springer-Verlag, 1990.

[Lee et al 1980] P.A. Lee, N. Ghani and K. Heron, "A recovery cache for the PDP-11," *IEEE Trans. Comput.*, vol. C-29, no. 6, pp.546-549, 1980.

[Lewerentz & Lindner 1995] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study "Production Cell"*, LNCS-891, Springer, Jan. 1995.

[Lötzbeyer 1996] A. Lötzbeyer, "Task description of a Fault-Tolerant Production Cell," Version 1.6, available from http://www.fzi.de/prost/projects/korsys/korsys.html, 1996.

[Lötzbeyer & Mühlfeld 1996] A. Lötzbeyer and R. Mühlfeld, "Task description of a flexible Production Cell with real time properties," FZI Technical Report, (ftp://ftp.fzi.de/pub/PROST/projects /korsys/task_descr_flex_2_2.ps), 1996.

[Lyu 1995] M.R. Lyu (ed.). *Software Fault Tolerance*, John Wiley & Sons, 1995.

[Parrington et al 1995] G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M.C. Little, "The design and implementation of Arjuna," *USENIX Comput. Syst. Journal*, vol. 8, no. 3, 1995.

[Powell 1991] D. Powell (Ed.). *Delta-4: a generic architecture for dependable distributed computing*, Springer (Berlin), 1991.

[Randell 1975] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Soft. Eng.*, vol. SE-1, no. 2, pp.220-232, 1975.

[Romanovsky et al 1998] A. Romanovsky, J. Xu and B. Randell, "Exception handling and coordinated atomic actions in object-oriented real-time distributed systems," in *1st Int. Symp. Object-Oriented Real-Time Distrib. Comput.*, pp.32-42, Kyoto, April 1998.

[Rubira & Stroud 1994] C.M.F. Rubira and R.J. Stroud, "Forward and backward error recovery in C++," *Object-Oriented Syst.*, vol. 1, no. 1, pp.61-85, 1994.

[Tso & Shokri 1996] K.S. Tso and E.H. Shokri, "An integrated environment for development and testing of software fault tolerance systems," in *Int. Workshop CAD, Test & Evaluation for Dependability (CADTED'96)*, pp.66-71, Beijing, 1996.

[Wing 1993] J.M. Wing, "Decomposing and recomposing transactional concepts," in *Object-Based Distributed Programming – ECOOP'93 Workshop* (eds. R. Guerraoui et al), Springer-Verlag, pp.111-121, 1993.

[Xu et al 1995a] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu, "Fault tolerance in concurrent object-oriented software through coordinated error recovery," in *25th Int. Symp. Fault-Tolerant Comput.*, pp.499-508, Pasadena, 1995.

[Xu et al 1995b] J. Xu, B. Randell, C.M.F. Rubira-Casavara and R.J. Stroud, "Toward an object-oriented approach to software fault tolerance," in *Recent Advances in Fault-Tolerant Parallel and Distributed Systems* (eds. D.K. Pradhan and D.R. Avresky), IEEE CS Press, pp.226-233, 1995.

[Xu et al 1998] J. Xu, A. Romanovsky, and B. Randell, "Coordinated exception handling in distributed object systems: from model to system implementation," in *18th Int. Conf. Distrib. Comput. Syst.*, pp.26-29, Amsterdam, 1998.

[Xu et al 1999] J. Xu, B. Randell, A. Romanovsky, R.J. Stroud, A. Zorzo, E. Canver, and F. von Henke, "Rigorous development of a safety-critical system based on coordinated atomic actions," in *29th Int. Symp. Fault-Tolerant Comput.*, Madison, pp.68-75, June 1999.

[Zorzo et al 1999] A.F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R.J. Stroud and I.S. Welch, "Using coordinated atomic actions to design safety-critical systems: a production cell case study," *Software – Practice & Experience*, vol. 29, no. 28, pp.677-697, 1999.

Appendix One: A reflective implementation

Following the reference architecture introduced in Section 3, it is quite straightforward to implement this pattern based on inheritance and delegation. However, our experience shows that a reflective implementation is also feasible when using a similar multi-level architecture. The meta level and base level can be treated as two separate levels in our reference architecture, each of which provides its own interface. For example, the base-level specifies the user interface for exploiting application functionality, and the meta-level defines the interface and components that determine the fault-tolerant behaviour of the application.

The proposed reference architecture is organized in the form of a multi-level system. This facilitates the transformation of the original architecture into different implementations such as a library-based system using inheritance and delegation or a reflective system. For example, in a reflective implementation the application level may be regarded as the base level, and the system support level may be defined as the meta level. At the meta level, all the control mechanisms become metaobjects and may be organized as several libraries. Similar to the generic FT interface components defined in the original multi-level architecture, a special metaobject protocol (MOP) must be implemented. This protocol serves as an interface to the meta level, and makes metaobjects accessible from base level in a well-defined and controlled manner. If necessary, the low system level may be implemented as a further meta level, or in other words, a meta meta level. In general, a reflective software system can have an infinite number of meta levels in which each meta level is controlled by a higher one, and where each meta level has its own metaobject protocol or MOP. In practice, most existing reflective languages or systems comprise only one or two meta levels. Figure A1 illustrates a reflective implementation in the distributed environment.

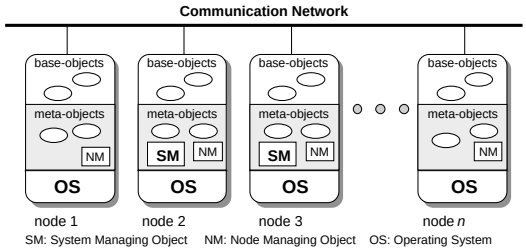

Figure A1 Reflective architecture in a distributed environment

Fault-tolerant objects in the original reference architecture, including software variants and the adjudicator, have been written in C++ and implemented at the base level. Various control mechanisms for different software fault tolerance schemes have been implemented at the meta level in Open C++. (Open C++ [Chiba & Masuda 1993] is a reflective version of C++ that provides the programmer with two levels of abstraction: the base level, like traditional C++ object-oriented programming; and the meta level which allows certain aspects of C++ to be redefined. In Open C++ operation calls to base level objects can be *intercepted* at the meta level by metaobjects.)

A major advantage of this reflective implementation is that for a given fault-tolerant object, the software fault tolerance scheme associated with it can be changed dynamically (even at run time) by changing and composing different control mechanisms at the meta level. When such dynamic changes are often desirable for certain applications, a reflective implementation minimizes the possible impact of the changes on application programs at the base level. However, a reflective implementation adds overheads to applications because of extra-level computation and indirection mechanisms. Its effectiveness also depends upon reflective facilities provided by a target implementation language.

We have conducted a set of experiments to assess the different run-time overheads imposed by different implementation approaches. In particular, the experiment sets have been re-run on four SUN workstations in an experimental distributed environment. Table 1 summarizes run-time overheads introduced by reflective operation calls. Normal C++ operation calls at the base level take from 3 to 64 microseconds, depending upon the location of different computing nodes. The corresponding operation calls in a reflective scheme (supported by Open C++) take from 6 to 100 microseconds. The overhead ratio is about 156% to 200%. However, the overhead imposed by reflective operation calls is insignificant in comparison with the entire cost introduced by fault tolerance schemes in both a library-based approach and a reflective approach. Moreover it can be seen to contribute an even smaller part to the whole overhead if the communication cost is taken into account. (These performance figures obtained from our experiments [Xu et al 1996] are very similar to the data generated subsequently from the FRIENDS system developed at LAAS [Fabre & Pérennou 1998].)

	Workstation One	Workstation Two	Workstation Three	Workstation Four
Normal operation call	56	3	64	8
Reflective operation call	100	6	100	16
Ratio	1.78	2	1.56	2

Table 1 Run-time overheads imposed by reflective operation calls [Xu et al 1996]

In a normal multi-level architecture, every level usually builds upon the levels below. There are no obvious bi-directional dependencies between two levels. In contrast with this, there are some mutual dependencies between levels in a reflective implementation. The base level builds on the meta level and vice-versa. For example, metaobjects can implement exceptional behaviour in case of an exception. However, this kind of exception handling must react according to the current state of computation. The meta level needs to retrieve the information from the base level, often from different components to those providing the interrupted service.

A reflective implementation can provide a greater degree of transparency than an implementation based on inheritance and delegation. It supports the change of fault tolerance schemes, statically or dynamically, even without modifying any source code of the fault-tolerant objects using the schemes. However, the reflective implementation is less efficient because of the complex relationship between the base level and the meta-level. Reflective capabilities often require extra processing including information retrieval, changing metaobjects, consistency checking, and communication between the two levels. All of these impose some performance penalty.

Appendix Two: Sample code

Controlling the execution of software variants. Our control structure is based on the Composite pattern, as shown in Figure 9. (The Backup pattern in [Subramanian & Tsai 1995] uses a similar pattern to implement RB; but our implementation is aimed at more complex concurrent and distributed systems.) The DR controls the execution of several variants by sending requests to class variant. The variant class is actually an abstract class that provides a common interface for a set of concrete variants that perform the operations requested. Apart from *n* subclasses implementing software variants, an additional subclass, called `Controller`, is organized as an aggregate of those variants and performs the actual control operations. The `Controller` has to know how many and which variants are needed for a particular request and how they are executed (e.g. sequentially, adaptively, or in parallel). To control the execution of variants, a `Controller` object has to create and store a list of concrete variant objects of its sibling classes.

According to the requests received, a `DR` object decides how many and which variants are needed and then adds the chosen variants to the `Controller` object. An `add()` operation must be declared in the variant abstract class in order to allow the `DR` object to add the variants to the `Controller`. This `add()` operation must be defined in the Controller subclass, but may not in the other variant subclasses if they do not support the addition operation. The sample code in C++ is as follows:

```
Class DR: public FT-m-controller {
public:
  DR( ){
      createVariant( );       //constructor for DR
  }
private:
  Variant *variant;
  void createVariant( );      //create and initialize variants
};
void DR :: createVariant( ) {
  variant = new Controller;  //create a Controller object
  variant.add(new variant1); //add all the variants chosen
  variant.add(new variant2);
  … … … …
  variant.add(new variantN);
}
```

Program 1 The `DR` class

For the DR scheme the order of addition determines the order in which variants are tried. Because the subsequent variants are executed only if their predecessors fail to satisfy the acceptance test, there is a need to maintain information about which variants have failed and which one should be tried next. This can be done by maintaining the state of the current variant being executed, e.g. using a variable called `currVariant`. The DR class has a private `variant` instance variable. The `createVariant` operation can initialize the variable to an instance of any variants if fault tolerance is not required (e.g. `variant = new variant1;`) or to an instance of the Controller class for the purpose of software fault tolerance. The Controller class may be defined as follows:

```
Class Controller: public Variant {
public:
  Controller( );
  void add(Variant*);
  void undo( );
  int action(externalObject);        //application-specific operation
private:
  int accepTest( );                  //acceptance test
  List<Variant*> variants;
  Variant* currVariant;
};
```

Program 2 The `Controller` class

The `Controller` constructor has to initialize a list of variants, i.e. the instance member `variants`, and initialize the current variant pointer: `currVariant = variants.first( )`. The `add` operation can be used by an AR object to create and add the variant objects in order:

```
Controller :: add(Variant* v) {
  variants.append(v);
}
```

Program 3 The `add` operation

A pointer to the current variant is maintained in the `Controller` and it points to the first member of the list initially, as shown in the code of the constructor. The variants in the list will be executed in order until one variant passes the acceptance test. The `Controller` implements the control in the following form:

```
int Controller :: action( ) {
  while(accepTest(currVariant.action( )) == ERROR) {
  undo( );
  currVariant = variants.next( );
  … … … …          //execute next variant
  }
}
```

Program 4 Implementation of the actual control

Finally, the `undo()` operation can be implemented as a virtual function which is bound to certain low-level checkpointing services.

Reference A

[Chiba & Masuda 1993] S. Chiba and T. Masuda, "Designing an extensible distributed language with a meta-level architecture," in *ECOOP'93*, pp.482-501, 1993.

[Fabre & Pérennou 1998] J-C. Fabre and T. Pérennou, "A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach," *IEEE Trans. Comput., Special Issue of Dependability of Computing Systems*, pp.78-95, 1998.

[Subramanian & Tsai 1996] S. Subramanian and W. Tsai, "Backup pattern: designing redundancy in object-oriented software," in *Pattern Languages of Program Design*, (eds. J. Coplien & D.C. Schnidt), Addision-Wesley, 1996.

[Xu et al 1996] J. Xu, B. Randell and A.F. Zorzo, "Implementing software-fault tolerance in C++ and Open C++," in *Int. Workshop CAD, Test & Evaluation for Dependability (CADTED'96)*, pp.224-229, Beijing, 1996.